MySQL 索引原理详解

1. 索引是什么？

1.1. 索引是什么

当一张表有 500 万条数据，在没有索引的 name 字段上执行一个查询：

```
select * from user_innodb where name ='青山';
```

如果 name 字段上面有索引呢?

```
ALTER TABLE user_innodb DROP INDEX idx_name;
ALTER TABLE user_innodb ADD INDEX idx_name (name);
```

索引的创建是需要消耗时间的。

有索引的查询和没有索引的查询相比，效率相差几十倍。

索引到底是什么呢？为什么可以对我们的查询产生这么大的影响？创建索引的时候做了什么事情?

1.1.1. 索引图解

> 数据库索引，是数据库管理系统（DBMS）中一个排序的数据结构，以协助快速查询、更新数据库表中数据。

数据是以文件的形式存放在磁盘上面的，每一行数据都有它的磁盘地址。如果没有索引的话，我们要从 500 万行数据里面检索一条数据，只能依次遍历这张表的全部数据（循环调用存储引擎的读取下一行数据的接口），直到找到这条数据。

但是我们有了索引之后，只需要在索引里面去检索这条数据就行了，因为它是一种特殊的专门用来快速检索的数据结构，我们找到数据存放的磁盘地址以后，就可以拿到数据了。

这个很容易理解，就像我们从一本 500 页的书里面去找特定的一小节的内容，肯定不可能从第一页开始翻。

这本书会有专门的目录，它可能只有几页的内容，它是按页码来组织的，可以根据拼音或者偏旁部首来查找，我们只要确定内容对应的页码，就能很快地找到我们想要的内容。

1.1.2. 索引类型

那在数据表上面，怎么创建一个索引？建表的时候指定，或者 alter table，也可以使用工具。

第一个是索引的名称，第二个是索引的列，比如我们是要对 id 创建索引还是对 name 创建索引。后面两个很重要，一个叫索引类型。

在 InnoDB 中，索引类型有三种，普通索引、唯一索引（主键索引是特殊的唯一索引）、全文索引。

普通（Normal）：也叫非唯一索引，是最普通的索引，没有任何的限制。

唯一（Unique）：唯一索引要求键值不能重复。另外需要注意的是，主键索引是一种特殊的唯一索引，它还多了一个限制条件，要求键值不能为空。主键索引用 primay key 创建。

全文（Fulltext）：针对比较大的数据，比如我们存放的是消息内容，有几 KB 的数据的这种情况，如果要解决 like 查询效率低的问题，可以创建全文索引。只有文本类型的字段才可以创建全文索引，比如 char、varchar、text。

```
create table m3 (
name varchar(50),
fulltext index(name)
);
```

```
select * from fulltext_test where match(content) against('咕泡学院' IN NATURAL LANGUAGE MODE);
```

在 5.6 的版本之后，MyISAM 和 InnoDB 都支持全文索引。但是 MySQL 自带的全文索引功能使用限制还是比较多，建议用其他的搜索引擎方案。

我们说索引是一种数据结构，那么它到底应该选择一种什么数据结构，才能实现数据的高效检索呢？

2. 索引存储模型推演

2.1. 二分查找

《幸运 52》有一个猜价格的游戏，限定时间之内猜中了就可以带回家。

在你报出价格之后主持人会告诉你低了还是高了。

如果价格在 10000-30000 之间，你会先猜多少？

这个就是二分查找的一种思想，也叫折半查找，每一次，我们都把候选数据缩小了一半。如果数据已经排过序的话，这种方式效率比较高。

所以第一个，可以考虑用有序数组作为索引的数据结构。

有序数组的等值查询和比较查询效率非常高，但是更新数据的时候会出现一个问题，可能要挪动大量的数据（改变 index），所以只适合存储静态的数据。

为了支持频繁的修改，比如插入数据，我们需要采用链表。链表的话，如果是单链表，它的查找效率还是不够高。

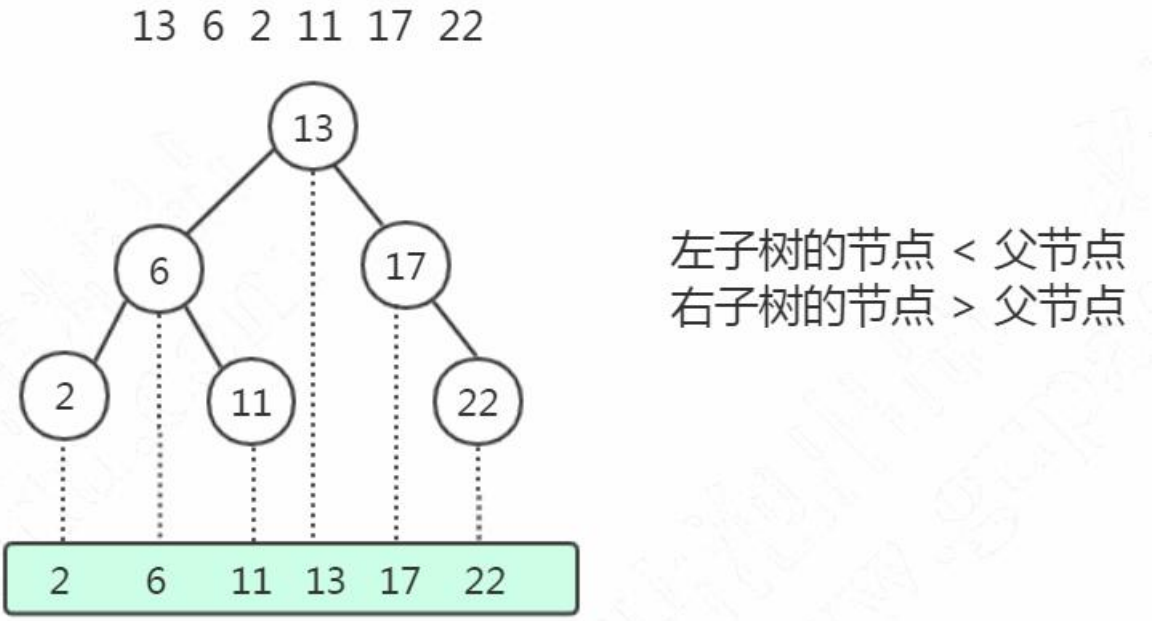
所以，有没有可以使用二分查找的链表呢？

为了解决这个问题，BST（Binary Search Tree）也就是我们所说的二叉查找树诞生了。

2.2. 二叉查找树（BST Binary Search Tree）

二叉查找树的特点是什么？

左子树所有的节点都小于父节点，右子树所有的节点都大于父节点。投影到平面以后，就是一个有序的线性表。



二叉查找树既能够实现快速查找，又能够实现快速插入。

但是二叉查找树有一个问题：

就是它的查找耗时是和这棵树的深度相关的，在最坏的情况下时间复杂度会退化成O(n)。

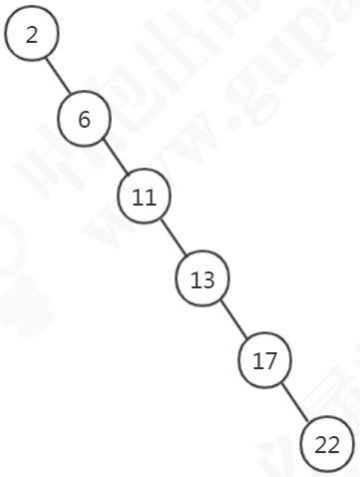
什么情况是最坏的情况呢?

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

还是刚才的这一批数字，如果我们插入的数据刚好是有序的，2、6、11、13、17、22。

这个时候二叉查找树变成了什么样了呢?

它会变成链表（我们把这种树叫做“斜树”），这种情况下不能达到加快检索速度的目的，和顺序查找效率是没有区别的。



造成它倾斜的原因是什么呢?

因为左右子树深度差太大，这棵树的左子树根本没有节点——也就是它不够平衡。

所以，有没有左右子树深度相差不是那么大，更加平衡的树呢?

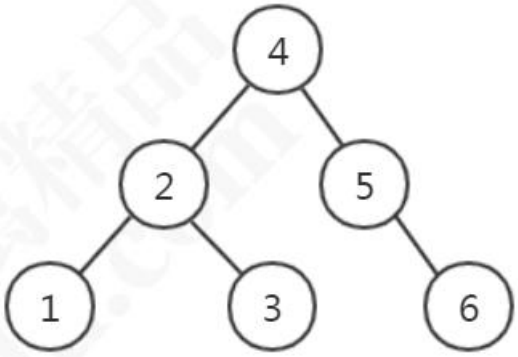
这个就是平衡二叉树，叫做 Balanced binary search trees，或者 AVL 树（AVL 是发明这个数据结构的人的名字缩写）。

2.3. 平衡二叉树（AVL Tree）（左旋、右旋）

平衡二叉树的定义：左右子树深度差绝对值不能超过 1。

比如左子树的深度是 2，右子树的深度只能是 1 或者 3。

这个时候我们再按顺序插入 1、2、3、4、5、6，一定是这样，不会变成一棵“斜树”。



那它的平衡是怎么做到的呢？怎么保证左右子树的深度差不能超过 1 呢？

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

插入 1、2、3。

注意看：当我们插入了 1、2 之后，如果按照二叉查找树的定义，3 肯定是要在 2 的右边的，这个时候根节点 1 的右节点深度会变成 2，但是左节点的深度是 0，因为它没有子节点，所以就会违反平衡二叉树的定义。

那应该怎么办呢？因为它是右节点下面接一个右节点，右-右型，所以这个时候我们要把 2 提上去，这个操作叫做左旋。

同样的，如果我们插入 7、6、5，这个时候会变成左左型，就会发生右旋操作，把 6 提上去。

所以为了保持平衡，AVL 树在插入和更新数据的时候执行了一系列的计算和调整的操作。

平衡的问题我们解决了，那么平衡二叉树作为索引怎么查询数据?

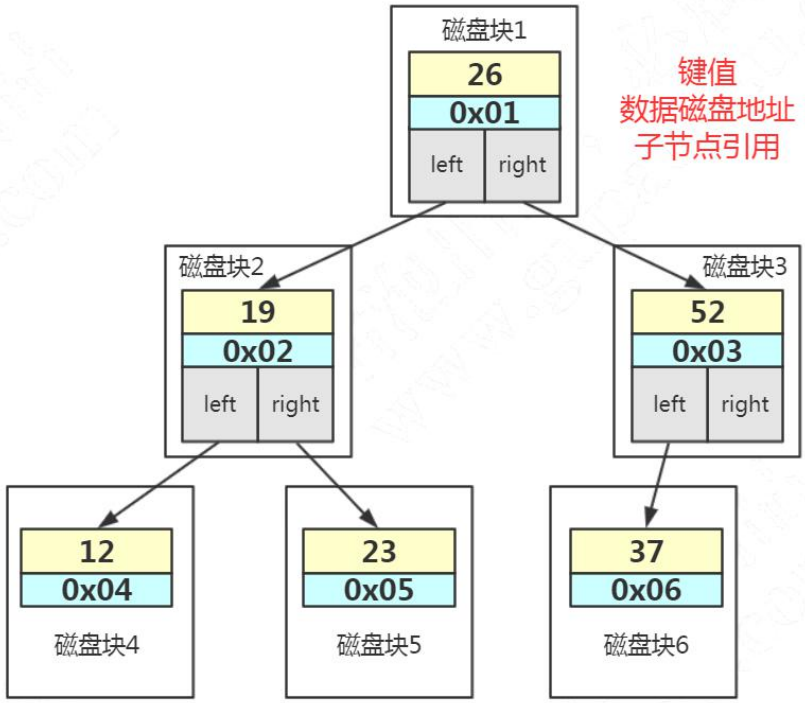
在平衡二叉树中，一个节点，它的大小是一个固定的单位，作为索引应该存储什么内容?

它应该存储三块的内容:

第一个是索引的键值。比如我们在 id 上面创建了一个索引，我在用 where id =1 的条件查询的时候就会找到索引里面的 id 的这个键值。

第二个是数据的磁盘地址，因为索引的作用就是去查找数据的存放的地址。

第三个，因为是二叉树，它必须还要有左子节点和右子节点的引用，这样我们才能找到下一个节点。比如大于 26 的时候，走右边，到下一个树的节点，继续判断。



如果是这样存储数据的话，我们来看一下会有什么问题。

首先，对于 InnoDB 来说，索引的数据，是放在硬盘上的。查看数据和索引的大小：

```
select
CONCAT(ROUND(SUM(DATA_LENGTH/1024/1024),2),'MB') AS data_len,
CONCAT(ROUND(SUM(INDEX_LENGTH/1024/1024),2),'MB') as index_len
from information_schema.TABLES
where table_schema='gupao' and table_name='user_innodb';
```

当我们用树的结构来存储索引的时候，因为拿到一块数据就要在 Server 层比较是不是需要的数据，如果不是的话就要再读一次磁盘。

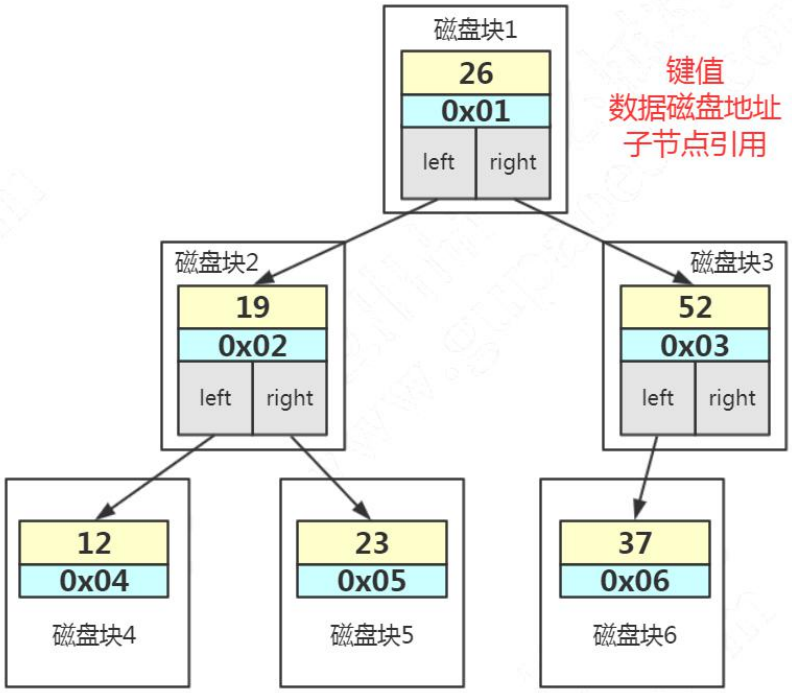
访问一个节点就要跟磁盘之间发生一次 I/O。InnoDB 操作磁盘的最小的单位是一页（或者叫一个磁盘块），大小是 16K(16384 字节)。

那么，一个树的节点就是 16K 的大小。

如果我们一个节点只存一个键值+数据+引用，例如整形的字段，可能只用了十几个或者几十个字节，它远远达不到 16384 字节的容量，所以访问一个树节点，进行一次 IO

的时候，浪费了大量的空间。

所以如果每个节点存储的数据太少，从索引中找到我们需要的数据，就要访问更多的节点，意味着跟磁盘交互次数就会过多，消耗的时间也越多。



比如上面这张图，我们一张表里面有 6 条数据，当我们查询 id=66 的时候，要查询两个子节点，就需要跟磁盘交互 3 次，如果我们有几百万的数据呢？这个时间更加难以估计。

所以解决方案是什么呢？

第一个，就是让每个节点存储更多的数据。

第二个，让每一层有更多的节点。也就是增加分叉数（或者路数），每个节点放更多的指针。

这样的话，就会极大地降低树的深度。我们的树就从原来的高瘦高瘦的样子，变成了矮胖矮胖的样子。

这个时候，我们的树就不再是二叉了，而是多叉，或者叫做多路。

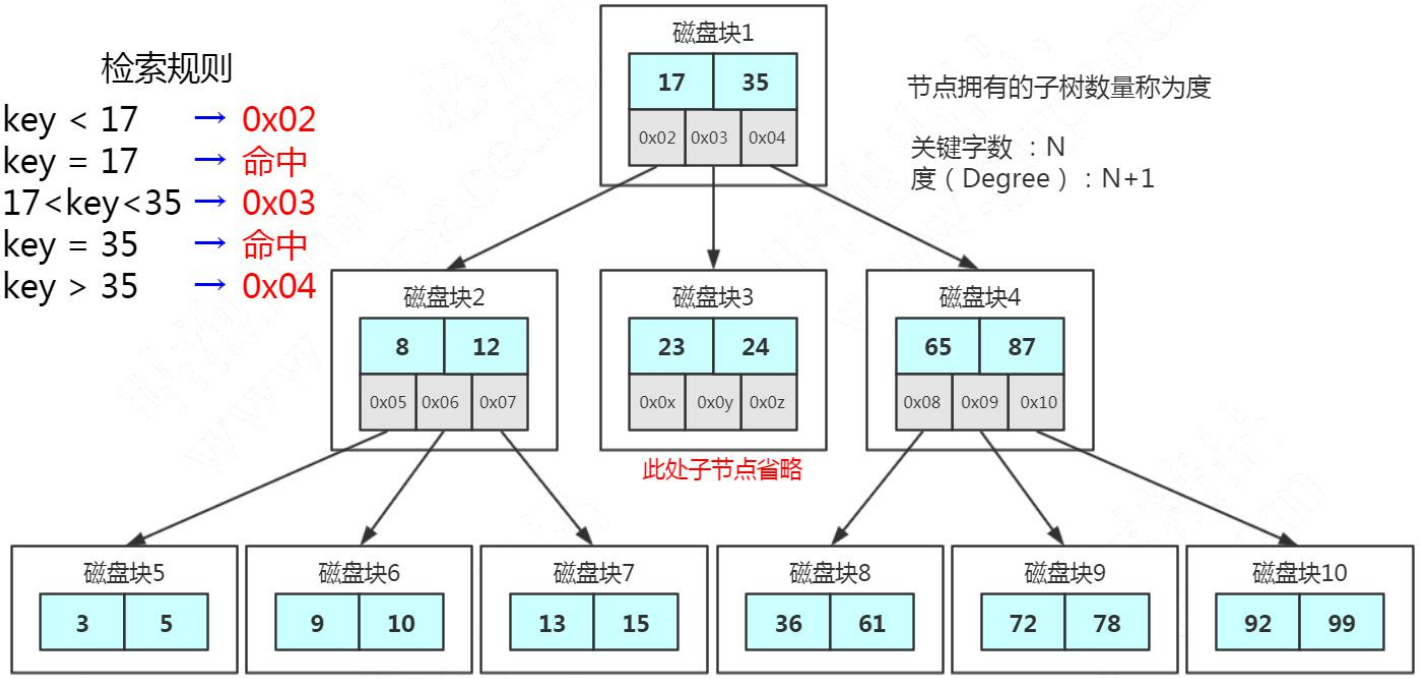
2.4. 多路平衡查找树（B Tree）（分裂、合并）

Balanced Tree

这个就是我们的多路平衡查找树，叫做 B Tree（B 代表平衡）。

跟 AVL 树一样，B 树在枝节点和叶子节点存储键值、数据地址、节点引用。

它有一个特点：分叉数（路数）永远比关键字数多 1。比如我们画的这棵树，每个节点存储两个关键字，那么就会有三个指针指向三个子节点。



B Tree 的查找规则是什么样的呢？

比如我们要在这张表里面查找 15。

因为 15 小于 17，走左边。

因为 15 大于 12，走右边。

在磁盘块 7 里面就找到了 15，只用了 3 次 IO。

这个是不是比 AVL 树效率更高呢？

那 B Tree 又是怎么实现一个节点存储多个关键字，还保持平衡的呢？跟 AVL 树有什么区别？

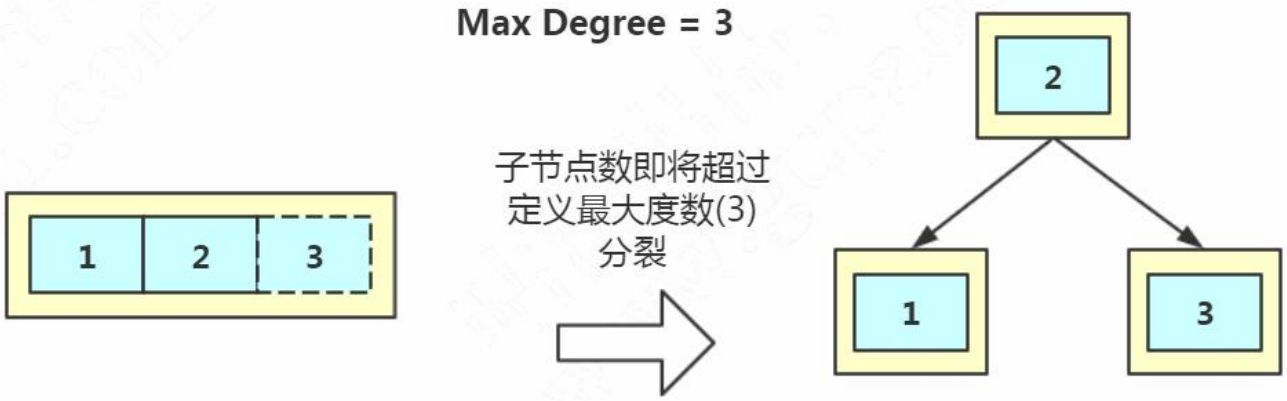
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

比如 Max Degree（路数）是 3 的时候，我们插入数据 1、2、3，在插入 3 的时候，本来应该在第一个磁盘块，但是如果一个节点有三个关键字的时候，意味着有 4 个指针，子节点会变成 4 路，所以这个时候必须进行分裂（其实就是 B+Tree）。把中间的数据 2 提上去，把 1 和 3 变成 2 的子节点。

如果删除节点，会有相反的合并的操作。

注意这里是分裂和合并，跟 AVL 树的左旋和右旋是不一样的。

我们继续插入 4 和 5，B Tree 又会出现分裂和合并的操作。



从这个里面也能看到，在更新索引的时候会有大量的索引的结构的调整，所以解释了为什么不要在频繁更新的列上建索引，或者为什么不要更新主键。

节点的分裂和合并，其实就是 InnoDB 页（page）的分裂和合并。

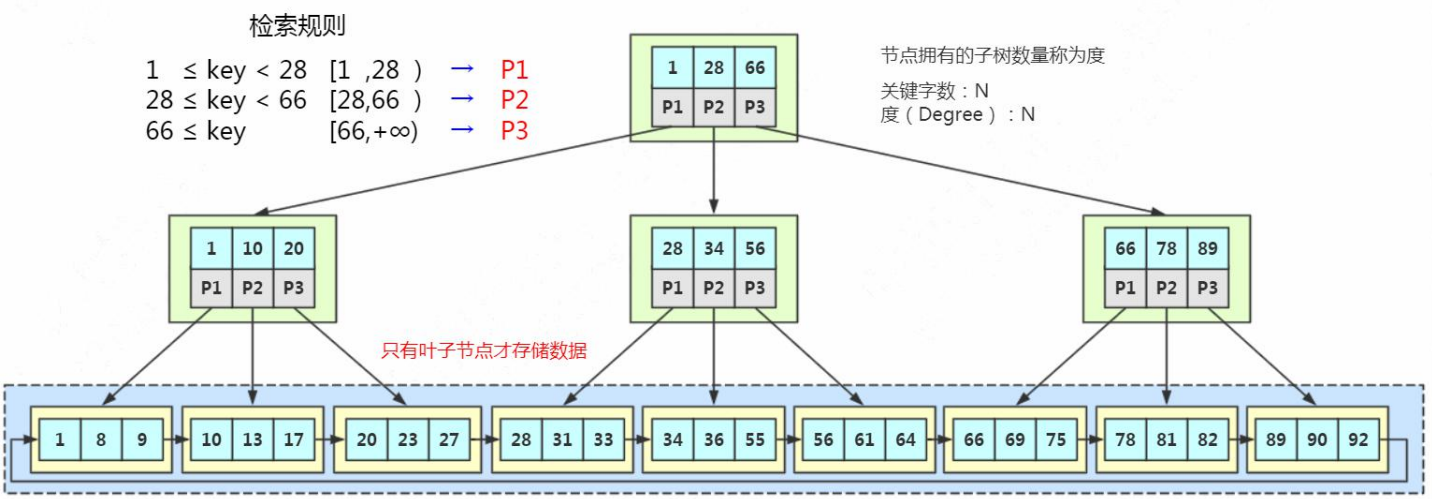
2.5. B+树（加强版多路平衡查找树）

B Tree 的效率已经很高了，为什么 MySQL 还要对 B Tree 进行改良，最终使用了 B+Tree 呢？

总体上来说，这个 B 树的改良版本解决的问题比 B Tree 更全面。

我们来看一下 InnoDB 里面的 B+树的存储结构：

【PPT 翻页】



MySQL 中的 B+Tree 有两个特点：

1、它的关键字的数量是跟路数相等的；

2、B+Tree 的根节点和枝节点中都不会存储数据，只有叶子节点才存储数据。

目前的认知：我们这要存放的数据是什么？是不是真实数据的地址？

搜索到关键字不会直接返回，会到最后一层的叶子节点。比如我们搜索 id=28，虽然在第一层直接命中了，但是数据地址在叶子节点上面，所以我还要继续往下搜索，一直到叶子节点。

3、B+Tree 的每个叶子节点增加了一个指向相邻叶子节点的指针，它的最后一个数据会指向下一个叶子节点的第一个数据，形成了一个有序链表的结构。

InnoDB 中的 B+Tree 这种特点带来的优势：

1)它是 B Tree 的变种，B Tree 能解决的问题，它都能解决。B Tree 解决的两大问题

是什么？（每个节点存储更多关键字；路数更多）

2)扫库、扫表能力更强（如果我们要对表进行全表扫描，只需要遍历叶子节点就可以了，不需要遍历整棵 B+Tree 拿到所有的数据）

3) B+Tree 的磁盘读写能力相对于 B Tree 来说更强（根节点和枝节点不保存数据区，所以一个节点可以保存更多的关键字，一次磁盘加载的关键字更多）

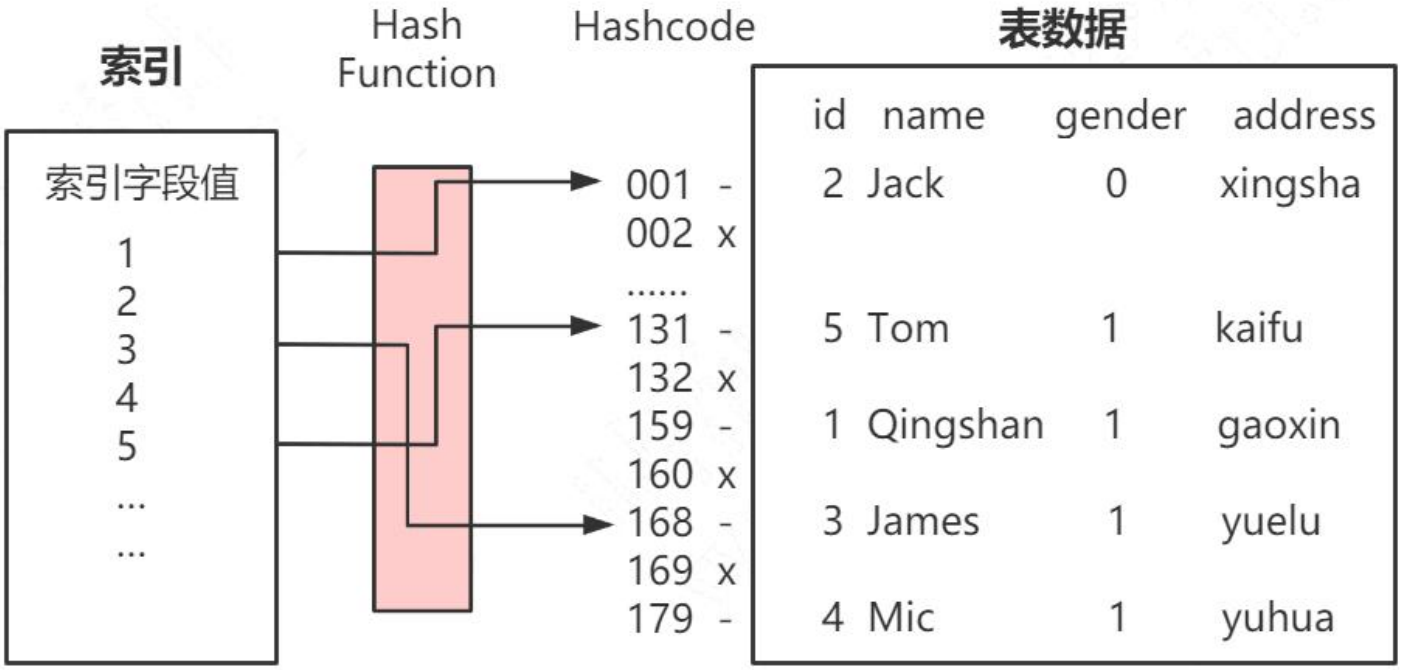
4)排序能力更强（因为叶子节点上有下一个数据区的指针，数据形成了链表）

5)效率更加稳定（B+Tree 永远是在叶子节点拿到数据，所以 IO 次数是稳定的）

2.6. 索引方式：真的是用的 B+Tree 吗？

在 Navicat 的工具中，创建索引，索引方式有两种。

HASH：以 KV 的形式检索数据，也就是说，它会根据索引字段生成哈希码和指针，指针指向数据。



哈希索引有什么特点呢？

第一个，它的时间复杂度是 O(1)，查询速度比较快。但是哈希索引里面的数据不是

按顺序存储的，所以不能用于排序。

第二个，我们在查询数据的时候要根据键值计算哈希码，所以它只能支持等值查询（= IN），不支持范围查询（> < >= <= between and）。

第三：如果字段重复值很多的时候，会出现大量的哈希冲突（采用拉链法解决），效率会降低。

需要注意的是，在 InnoDB 中，不能显示地创建一个哈希索引（所谓的支持哈希索引指的是 Adaptive Hash Index）。

https://dev.mysql.com/doc/refman/5.7/en/create-index.html

memory 存储引擎可以使用 Hash 索引。

```
CREATE TABLE `user_memory` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `gender` tinyint(1) DEFAULT NULL,
  `phone` varchar(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_name` (`name`) USING HASH
) ENGINE=MEMORY AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;
```

如果说面试的时候问到了为什么不用红黑树：

红黑树的种种约束保证的是什么？最长路径不超过最短路径的二倍。不太适合于数据库索引。适合内存的数据机构，例如实现一致性哈希。

因为 B Tree 和 B+Tree 的特性，它们广泛地用在文件系统和数据库中，例如 Windows 的 HPFS 文件系统，Oracel、MySQL、SQLServer 数据库。

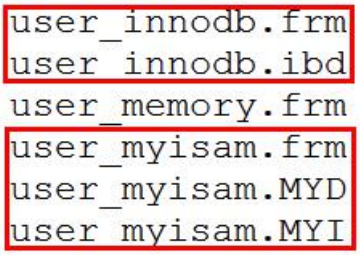
3. B+Tree 落地形式

3.1. MySQL 数据存储文件

上一节课我们知道了不同的存储引擎文件不一样。

```
show VARIABLES LIKE 'datadir';
```

每张 InnoDB 的表有两个文件（.frm 和.ibd），MyISAM 的表有三个文件（.frm、.MYD、.MYI）。



有一个是相同的文件，.frm。 .frm 是 MySQL 里面表结构定义的文件，不管你建表的时候选用任何一个存储引擎都会生成，我们就不看了。

我们主要看一下其他两个文件是怎么实现 MySQL 不同的存储引擎的索引的。

3.2.1.MyISAM

在 MyISAM 里面，另外有两个文件：

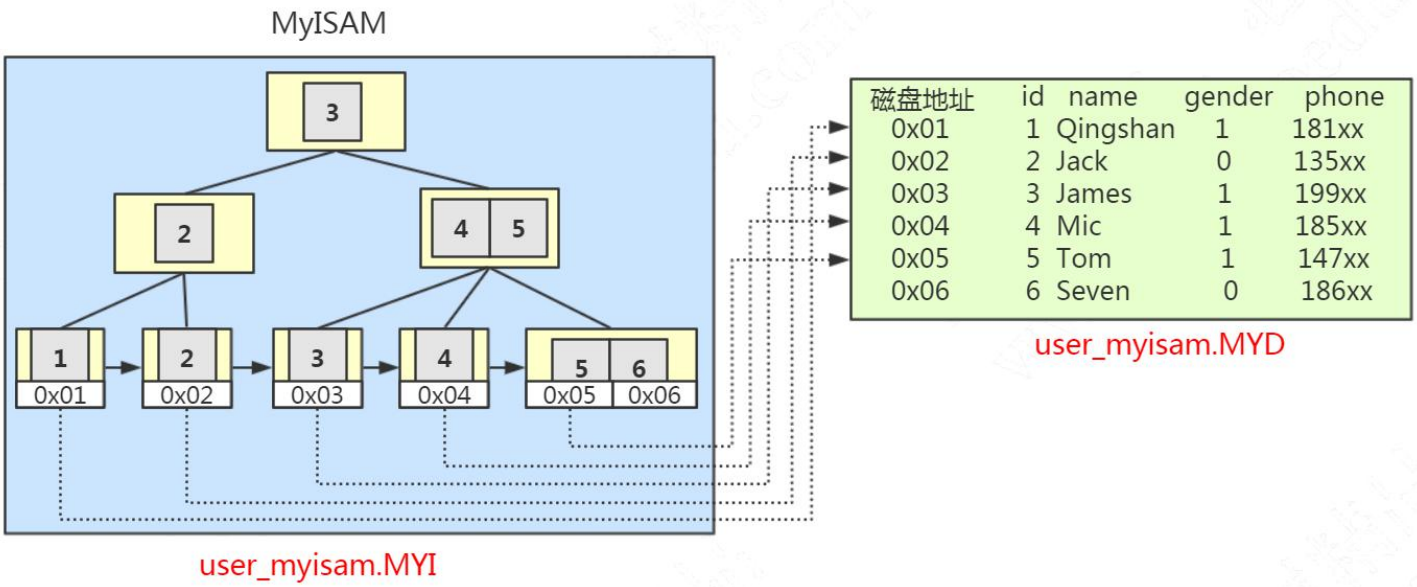
一个是.MYD 文件，D 代表 Data，是 MyISAM 的数据文件，存放数据记录，比如我们的 user_myisam 表的所有的表数据。

一个是.MYI 文件，I 代表 Index，是 MyISAM 的索引文件，存放索引，比如我们在 id 字段上面创建了一个主键索引，那么主键索引就是在这个索引文件里面。

也就是说，在 MyISAM 里面，索引和数据是两个独立的文件。

那我们怎么根据索引找到数据呢?

MyISAM 的 B+Tree 里面，叶子节点存储的是数据文件对应的磁盘地址。所以从索引文件.MYI 中找到键值后，会到数据文件.MYD 中获取相应的数据记录。

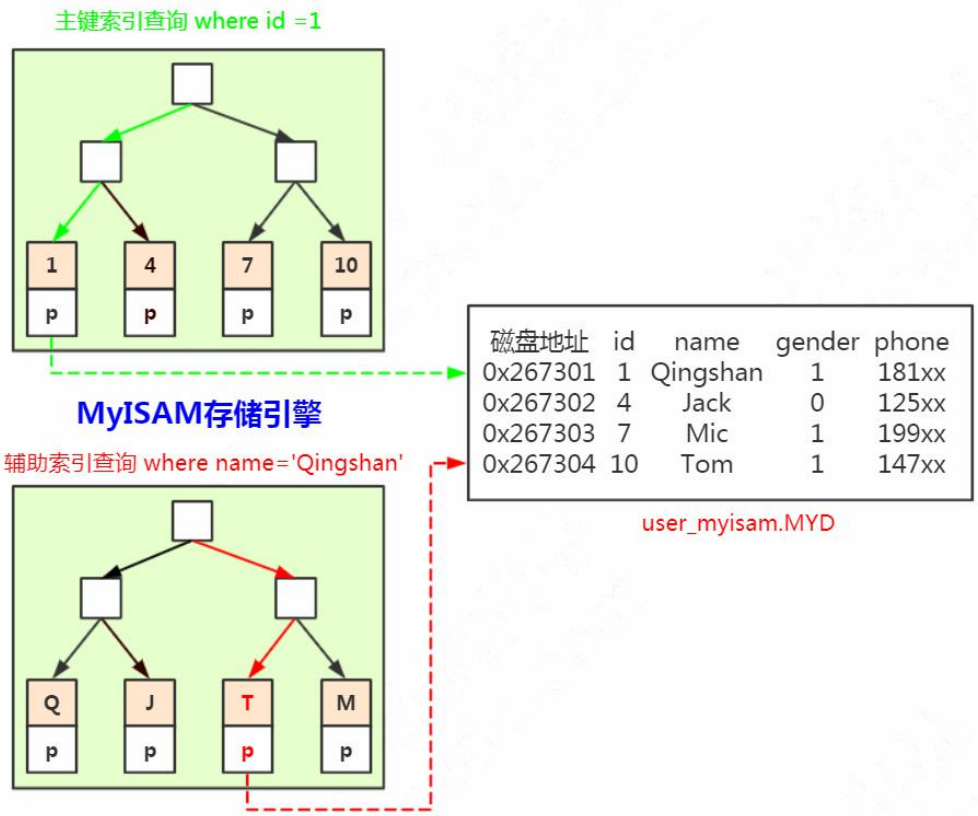


如果是辅助索引，有什么不一样?

```
ALTER TABLE user_innodb DROP INDEX index_user_name;
ALTER TABLE user_innodb ADD INDEX index_user_name (name);
```

在 MyISAM 里面，辅助索引也在这个.MYI 文件里面。

辅助索引跟主键索引存储和检索数据的方式是没有任何区别的，一样是在索引文件里面找到磁盘地址，然后到数据文件里面获取数据。



这个就是 MyISAM 里面的索引落地的形式。但是在 InnoDB 里面是不一样的。我们来看一下。

3.2.2. InnoDB

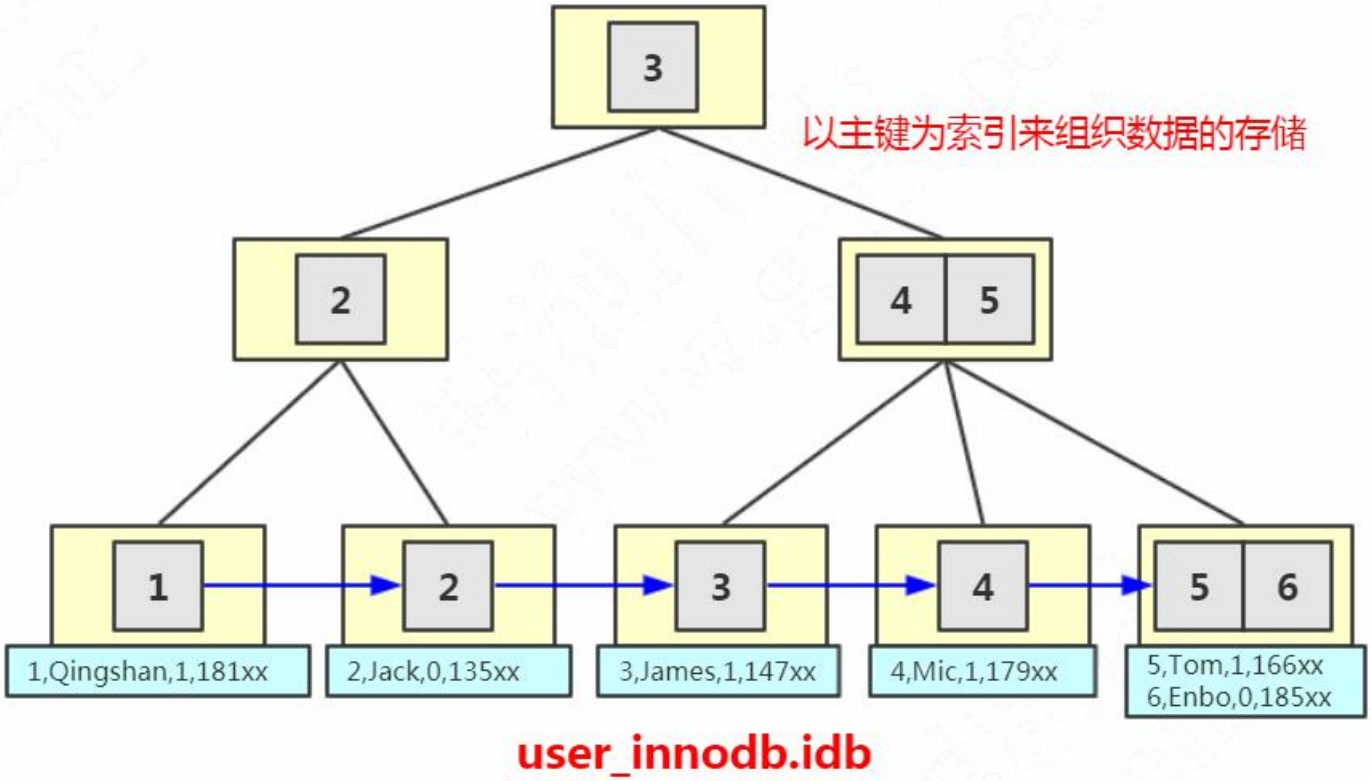
InnoDB 只有一个文件（.ibd 文件），那索引放在哪里呢?

在 InnoDB 里面，它是以主键为索引来组织数据的存储的，所以索引文件和数据文件是同一个文件，都在.ibd 文件里面。

在 InnoDB 的主键索引的叶子节点上，它直接存储了我们的数据。

所以，为什么说在 InnoDB 中索引即数据，数据即索引，就是这个原因。

但是这里会有一个问题，一张 InnoDB 的表可能有很多个多索引，数据肯定是只有一份的，那数据在哪个索引的叶子节点上呢?

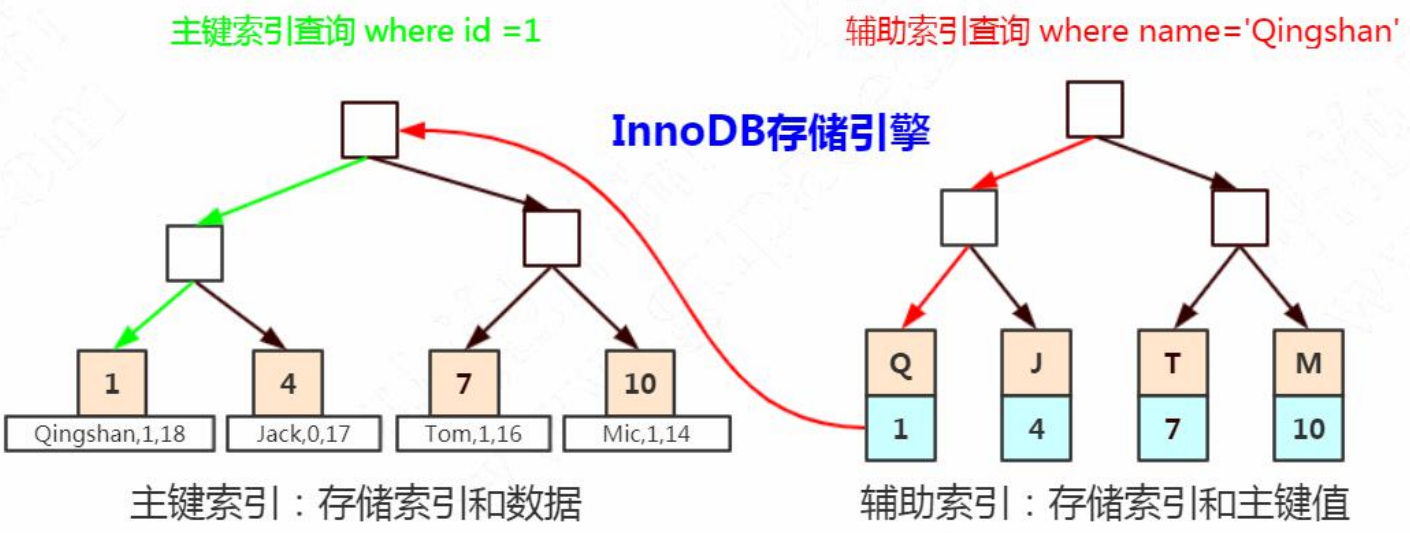


这里要给大家介绍一个叫做聚集索引（聚簇索引）的概念。

就是索引键值的逻辑顺序跟表数据行的物理存储顺序是一致的。（比如字典的目录是按拼音排序的，内容也是按拼音排序的，按拼音排序的这种目录就叫聚集索引）。

InnoDB 组织数据的方式就是(聚集)索引组织表(clustered index organize table)。如果说一张表创建了主键索引，那么这个主键索引就是聚集索引，决定数据行的物理存储顺序。

问题来了，那主键索引之外的索引，他们存储什么内容，他们的叶子节点上没有数据怎么检索完整数据？比如在 name 字段上面建的普通索引。



InnoDB 中，主键索引和辅助索引是有一个主次之分的。刚才我们讲了，如果有主键索引，那么主键索引就是聚集索引。其他的索引统一叫做“二级索引”或者辅助索引。

二级索引存储的是辅助索引的键值，例如在 name 上建立索引，节点上存的是 name 的值，qingshan mic tom 等等。

而二级索引的叶子节点存的是这条记录对应的主键的值。比如 qingshan id=1，jack id=4……

所以，二级索引检索数据的流程是这样的：

当我们用 name 索引查询一条记录，它会在二级索引的叶子节点找到 name=qingshan，拿到主键值，也就是 id=1，然后再到主键索引的叶子节点拿到数据。

从这个角度来说，因为主键索引比二级索引少扫描了一棵 B+Tree，它的速度相对会快一些。

但是，如果一张表没有主键怎么办？那完整的记录放在哪个索引的叶子节点？或者，这张表根本没有索引呢？数据放在哪里？

https://dev.mysql.com/doc/refman/5.7/en/innodb-index-types.html

1、如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引。

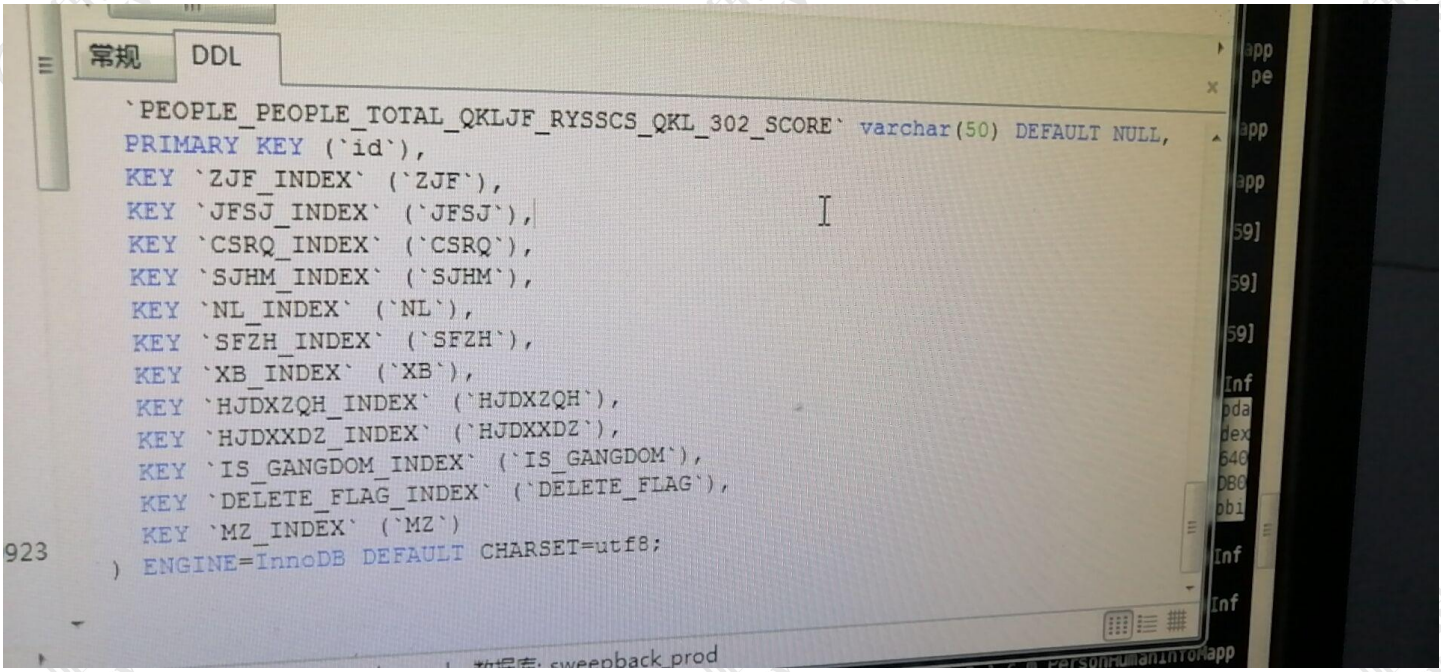
2、如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引作为主键索引。

3、如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐藏的聚集索引，它会随着行记录的写入而主键递增。

```
select _rowid name from t2;
```

4. 索引使用原则

我们容易有一个误区，就是在经常使用的查询条件上都建立索引，索引越多越好，那到底是不是这样呢?



4.1. 列的离散（sàn）度

第一个叫做列的离散度，我们先来看一下列的离散度的公式：

count(distinct(column_name)) : count(*)，列的全部不同值和所有数据行的比例。

数据行数相同的情况下，分子越大，列的离散度就越高。

id	name	gender	phone
1	青山	0	13101880079
2	郑娟	1	15501862216
3	王致蒽	0	18504734367
4	秦柜墨	1	15106797784
5	王涩鑫	0	15000770789
6	王锢	0	15900528227
7	朱癌	1	13806617196
8	陈怀密	0	13707077795
9	冯混钩	0	15604604290
10	蒋迥	0	13702963295

简单来说，如果列的重复值越多，离散度就越低，重复值越少，离散度就越高。

我们不建议大家在离散度低的字段上建立索引。

没有索引的时候查一遍：

```
SELECT * FROM `user_innodb` WHERE gender = 0;
```

建立索引之后再查一遍：

```
ALTER TABLE user_innodb DROP INDEX idx_user_gender;
ALTER TABLE user_innodb ADD INDEX idx_user_gender (gender);   -- 耗时比较久
SELECT * FROM `user_innodb` WHERE gender = 0;
```

发现消耗的时间更久了。

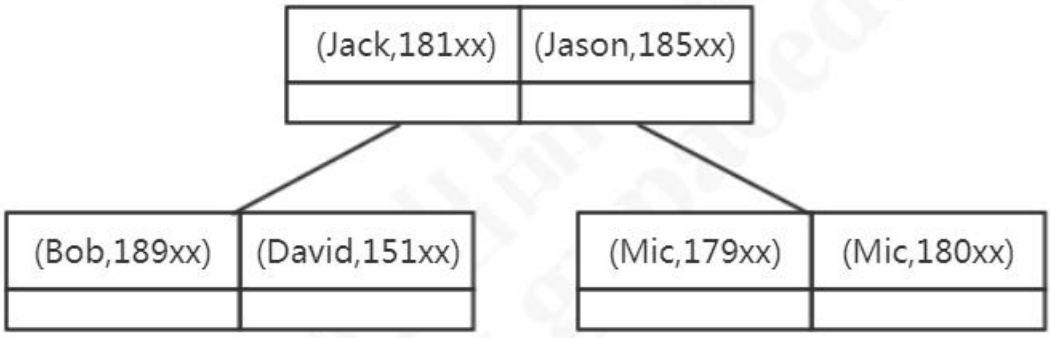
4.2. 联合索引最左匹配

前面我们说的都是针对单列创建的索引，但有的时候我们的多条件查询的时候，也会建立联合索引，举例：查询成绩的时候必须同时输入身份证和考号。

单列索引可以看成是特殊的联合索引。

比如我们在 user 表上面，给 name 和 phone 建立了一个联合索引。

```
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX comidx_name_phone (name,phone);
```



多个键值的B+Tree

联合索引在 B+Tree 中是复合的数据结构，它是按照从左到右的顺序来建立搜索树的（name 在左边，phone 在右边）。

从这张图可以看出来，name 是有序的，phone 是无序的。当 name 相等的时候，phone 才是有序的。

这个时候我们使用 where name= '青山' and phone = '136xx '去查询数据的时候，B+Tree 会优先比较 name 来确定下一步应该搜索的方向，往左还是往右。如果 name 相同的时候再比较 phone。但是如果查询条件没有 name，就不知道第一步应该查哪个节点，因为建立搜索树的时候 name 是第一个比较因子，所以用不到索引。

4.2.1. 什么时候用到联合索引

所以，我们在建立联合索引的时候，一定要把最常用的列放在最左边。

比如下面的三条语句，大家觉得用到联合索引了吗？

1）使用两个字段，用到联合索引：

```
EXPLAIN SELECT * FROM user_innodb WHERE name= '权亮' AND phone = '15204661800';
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	user_innodb	(Null)	ref	comidx_name_phone	comidx_name_phone	1070

2）使用左边的 name 字段，用到联合索引：

```
EXPLAIN SELECT * FROM user_innodb WHERE name= '权亮'
```

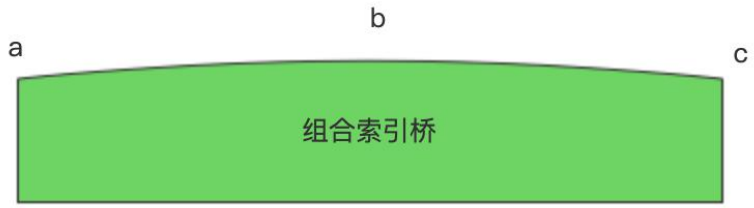
id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	user_innodb	(Null)	ref	comidx_name_phone	comidx_name_phone	1023	const

3）使用右边的 phone 字段，无法使用索引，全表扫描：

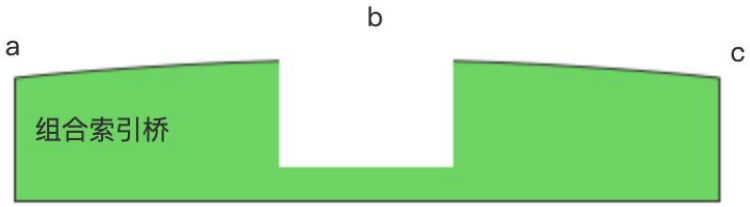
```
EXPLAIN SELECT * FROM user_innodb WHERE phone = '15204661800'
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	user_innodb	(Null)	ALL	(Null)	(Null)	(Null)	(Nul	996770

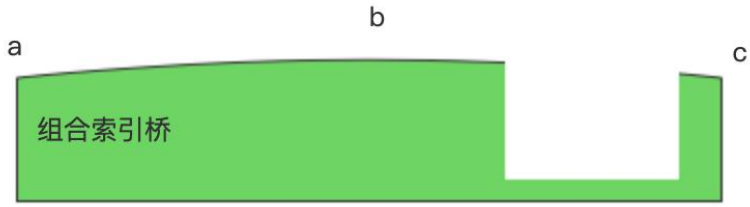
- where a and c and b：索引字段全部命中



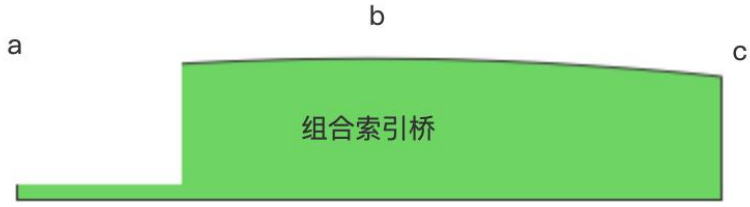
- where a and c：索引命中a



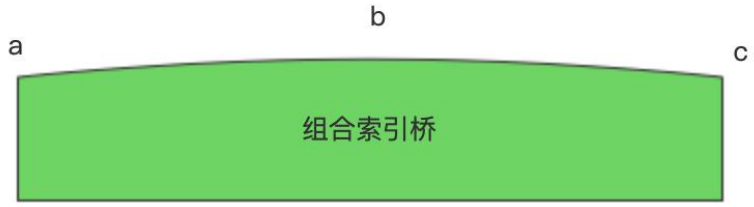
- where a and b：索引命中a、b



- where b and c：索引未命中



- where a and b order by c：索引字段全部命中



4.2.2. 如何创建联合索引

有一天我们的 DBA 找到我，说我们的项目里面有两个查询很慢，按照我们的想法，一个查询创建一个索引，所以我们针对这两条 SQL 创建了两个索引，这种做法觉得正确吗?

```
CREATE INDEX idx_name on user_innodb(name);
CREATE INDEX idx_name_phone on user_innodb(name,phone);
```

当我们创建一个联合索引的时候，按照最左匹配原则，用左边的字段 name 去查询的时候，也能用到索引，所以第一个索引完全没必要。

相当于建立了两个联合索引(name),(name,phone)。

如果我们创建三个字段的索引 index(a,b,c)，相当于创建三个索引：

index(a)

index(a,b)

index(a,b,c)

用 where b=？和 where b=? and c=？是不能使用到索引的。

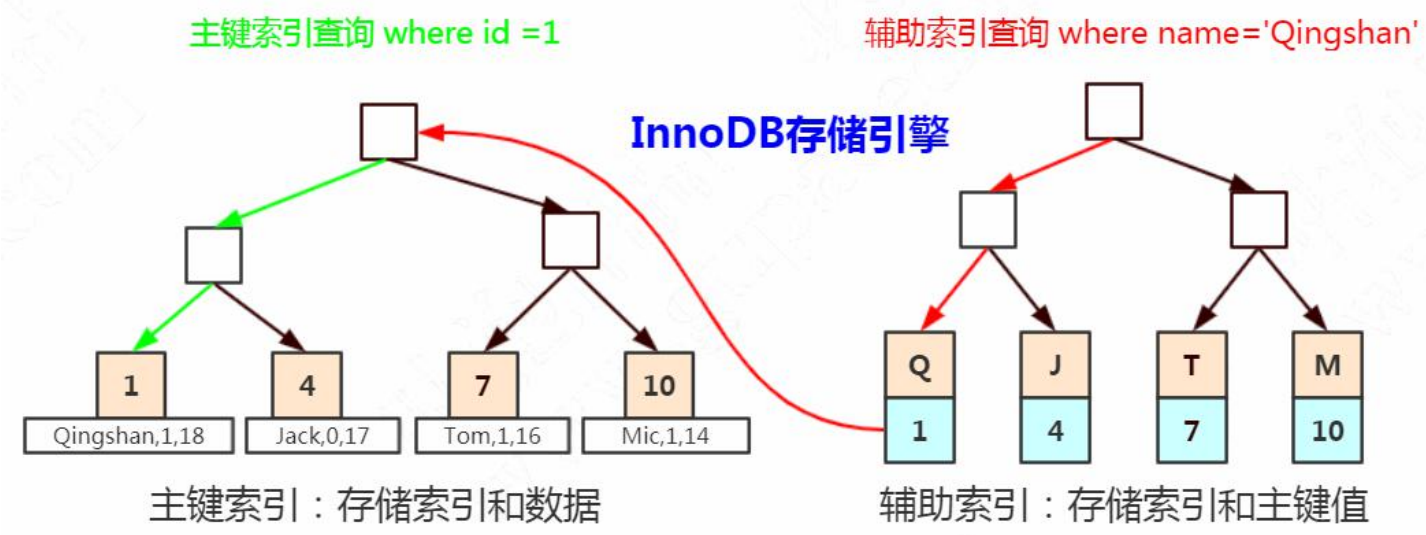
这里就是 MySQL 里面联合索引的最左匹配原则。

4.3. 覆盖索引

什么叫回表：

非主键索引，我们先通过索引找到主键索引的键值，再通过主键值查出索引里面没有的数据，它比基于主键索引的查询多扫描了一棵索引树，这个过程就叫回表。

例如：select * from user_innodb where name = '青山';



在辅助索引里面，不管是单列索引还是联合索引，如果 select 的数据列只用从索引中就能够取得，不必从数据区中读取，这时候使用的索引就叫做覆盖索引，这样就避免了回表。

Extra 里面值为“Using index”代表使用了覆盖索引。

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	user_innodb	ref	comidx_name_phone	comidx_name_phone	1070	const,const	1	100	Using where;Using index

我们先来创建一个联合索引：

```
-- 创建联合索引
ALTER TABLE user_innodb DROP INDEX comixd_name_phone;
ALTER TABLE user_innodb add INDEX `comixd_name_phone` (`name`,`phone`);
```

这三个查询语句都用到了覆盖索引：

```
EXPLAIN SELECT name,phone FROM user_innodb WHERE name= '青山' AND phone = ' 13666666666';
EXPLAIN SELECT nameFROM user_innodb WHERE name= '青山' AND phone = ' 13666666666';
EXPLAIN SELECT phone FROM user_innodb WHERE name= '青山' AND phone = ' 13666666666';
```

select * ，此处用不到覆盖索引。

如果改成只用 where phone = 查询呢？大家自己试试。按照我们之前的分析，它是用不到索引的。

实际上可以用到覆盖索引！覆盖索引跟是否可能使用索引没有直接关系。

很明显，因为覆盖索引减少了 IO 次数，减少了数据的访问量，可以大大地提升查询效率。

5.索引的创建与使用

因为索引对于改善查询性能的作用是巨大的，所以我们的目标是尽量使用索引。

5.1. 在什么字段上索引？

1、在用于 where 判断 order 排序和 join 的（on）字段上创建索引

2、索引的个数不要过多。

——浪费空间，更新变慢。

3、区分度低的字段，例如性别，不要建索引。

——离散度太低，导致扫描行数过多。

4、频繁更新的值，不要作为主键或者索引。

——页分裂

5、随机无序的值，不建议作为主键索引，例如身份证、UUID。

——无序，分裂

6、创建复合索引，而不是修改单列索引

5.2. 什么时候索引失效？

1、索引列上使用函数（replace\SUBSTR\CONCAT\sum count avg）、表达式计算（+ - * /）：https://www.runoob.com/mysql/mysql-functions.html

```
explain SELECT * FROM `t2` where id+1 = 4;
```

2、字符串不加引号，出现隐式转换

```
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX comidx_name_phone (name,phone);
```

```
explain SELECT * FROM `user_innodb` where name = 136;
explain SELECT * FROM `user_innodb` where name = '136';
```

3、like 条件中前面带%

where 条件中 like abc%，like %2673%，like %888 都用不到索引吗？为什么？

```
explain select *from user_innodb where name like 'wang%';
explain select *from user_innodb where name like '%wang';
```

过滤的开销太大。这个时候可以用全文索引。

4、负向查询

NOT LIKE 不能：

```
explain select *from employees where last_name not like 'wang'
```

!= （<>）和 NOT IN 在某些情况下可以：

```
explain select *from employees where emp_no not in (1)
explain select *from employees where emp_no <> 1
```

注意跟数据库版本、数据量、数据选择度都有关系。

其实，用不用索引，最终都是优化器说了算。

优化器是基于什么的优化器?

基于 cost 开销(Cost Base Optimizer)，它不是基于规则(Rule-Based Optimizer)，也不是基于语义。怎么样开销小就怎么来。

https://docs.oracle.com/cd/B10501_01/server.920/a96533/rbo.htm#38960
https://dev.mysql.com/doc/refman/5.7/en/cost-model.html

使用索引有基本原则，但是没有具体细则，没有什么情况一定用索引，什么情况一定不用索引的规则。

作者：咕泡学院-青山